The Use of Proof Planning for Cooperative Theorem Proving

The Use of Proof Planning for Cooperative Theorem Proving

Helen Lowe, Alan Bundy, & Duncan McLean

DAI Research Paper No. 745

April 4, 1995

Submitted to the Journal of Symbolic Computation

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

The Use of Proof Planning for Cooperative Theorem Proving

Helen Lowe, Alan Bundy, & Duncan McLean

Abstract

We describe BARNACLE: a cooperative interface to an inductive theorem prover. The cooperative nature of the BARNACLE interface is made possible by proof planning.

Proof planning is a technique for guiding the search for a proof in automatic theorem proving. Common patterns of reasoning in proofs are identified and represented computationally as *proof plans*. These proof plans are then used to guide the search for proofs of new conjectures. Where a proof requires more than common patterns of reasoning, proof planning needs to be supplemented by human interaction.

Proof planning makes new kinds of user interaction possible. Proof plans structure proofs hierarchically. This can be used to present partial proofs to users without overwhelming them with detail. Proof plans use a meta-logic to relate each chunk of a proof to its parents and daughters in the hierarchy and to its subparts. Proof plans sometimes annotate the proof steps to display the rationale behind them. The relations between proof chunks and the annotations can both be used to help users understand the state of proof attempts. This improved understanding can help them find patches to failed proofs and the meta-logic provides a high-level language for specifying the patch.

Key words and phrases. Proof Planning, Co-operative Theorem Proving, Explanations.

1 Introduction

MAPLE (Char, 1993) and Mathematica (Wolfram, 1991) represent notable success stories in the field of automated mathematical computation, and there has been similar success in providing sophisticated interfaces to such systems. For example, the MathCAD system available on PC, Apple Macintosh, and UNIX systems incorporates MAPLE for symbolic processing and hence provides a user-friendly environment for a host of MAPLE facilities including equation solving, differentiation, integration, and symbolic matrix operations. These all share in common the fact that algorithms exist for their complete automation leaving maybe only reformulation of problems and efficiency considerations as residual preoccupations of developers and users alike.

No such fail-safe algorithms can exist for general theorem proving, because it is an undecidable problem. Heuristic solutions are only partially successful. User intervention in some shape or form is likely to remain prominent in the foreseeable future. Interfaces to theorem provers must provide more than good display and editing facilities. They must explain the state of the proof, to assist the user in assisting the theorem prover. In other words, a dialogue is needed to facilitate a co-operative effort between human and automaton. This provides new challenges to the design and implementation of user interfaces to theorem proving systems.

2 Automated Theorem Proving and the Problem of Search

The major problem we face in automating theorem proving is controlling the search for a proof. It is relatively easy to formalise the rules of reasoning and represent them in a computer program.

*We thank Matthew Padfield for his work on coloured annotations, Ian Green for help with LaTeX, and both the Mathematical Reasoning Group at Edinburgh University and students at Napier University for much helpful discussion and feedback. The research reported in this paper was supported by SERC grant GR/H/23610.

A proof is a tree in which the leaves are labelled by axioms, the root by the conjecture and each arc by a rule. Theorem proving can then be automated by chaining rules together in a search for a proof tree. Unfortunately, at each node of the proof tree, many rules apply. To find a proof we must search. For non-trivial theorems this search becomes prohibitively expensive both in time and space. This phenomenon is called the *combinatorial explosion.*

Many solutions to the combinatorial explosion have been tried: the search process can be guided by heuristics which often pick the right rule; human interaction can be used to make crucial choices; and decision procedures and normal forms can be used for parts of the proof. One of the most successful approaches has been the use of *tactics* (Gordon *et al*, 1979). A tactic is a computer program whose result is the application of proof rules. Such a tactic might encode a decision procedure, a normal form or some other frequently useful combination of proof rules. Tactics are often used in combination with human interaction; the human can choose to apply a primitive rule or a tactic which encodes many rules. A theorem prover which supports this use of tactics is often called a *proof editor.* The user is seen as editing a partial proof by filling gaps in it with rule and tactic applications.

3 Proof Planning

In the mathematical reasoning group at Edinburgh we have developed the use of tactics into what we call *proof planning* (Bundy, 1988). A *proof plan* is the outline or global structure of a proof. We have observed that many proofs share the same overall proof plan or contain common patterns of reasoning. We have studied families of similar proofs and tried to extract these common patterns and represent them computationally. They can then be used to guide the search for proofs of new conjectures from the same family. In particular, we have looked at many proofs by mathematical induction, especially those that arise in proofs about properties of recursive programs, and extracted the common patterns from them. A lot of this analysis was informed by the earlier work of Boyer and Moore (Boyer & Moore, 1979).

Frequently occurring patterns of reasoning can be represented as tactics and used to drive a proof editor. These tactics can often be arranged in hierarchies, with larger tactics being composed of smaller ones. Despite these common patterns, there is some variation between proofs from the same family, and we must also account for this. One solution is to allow a human user to put together the particular combination of tactics needed for the current conjecture. Using proof planning, we can automate much of this process.

To each general-purpose tactic we associate a specification of its behaviour. Our proof planner, *CLAM* (Bundy *et al*, 1990), then reasons with this specification to custom build a special-purpose tactic for the current conjecture. The specification consists of the preconditions under which the tactic is applicable and the effects of its application. *CLAM* links the effects of earlier tactics to the preconditions of later ones, thus chaining the general-purpose tactics together into a special-purpose tactic for the whole proof. Preconditions and effects both describe syntactic properties of the expressions being manipulated and are expressed in a *meta-logic, i.e.* a logic for reasoning about logics. *CLAM* uses a formal, computational representation of this meta-logic, but it can also be rendered into English to provide explanations to users about the role of each tactic within the proof plan.

4 Rippling

Rippling is a key tactic in inductive proofs. Not only does it control one of the most important parts of the proof, but many of the other tactics can be seen either as preparing for it or as completing its work. We shall use it to illustrate the use of tactics and proof planning.

An inductive proof consists of base and step cases. In a base case the theorem is proved for a boundary value, *e.g.* 0. In a step case we assume the theorem for one value, *e.g.* n, and

prove it for the next, *e.g.* $s(n)$[1]. The step case assumption is called the *induction hypothesis* and the goal is the *induction conclusion*. The induction hypothesis and induction conclusion are necessarily syntactically similar, *e.g.* they may differ only by the presence of various $s(\ldots)$s. The bulk of the step case usually consists of the manipulation of the induction conclusion until parts of it match the induction hypothesis. These parts can then be assumed true, leaving a residual expression which is easier to prove. Following Boyer and Moore, we call the tactic that uses the induction hypothesis in this way: *fertilization*. Rippling is the tactic that prepares for fertilization by manipulating the induction conclusion.

The idea of rippling is to identify the bits of the induction conclusion that already match the induction hypothesis (called the *skeleton*) and the bits by which they differ (called the *wave-fronts*). Special annotations are used to label the different bits. The wave-fronts are then moved, in a series of rewritings, to places where they no longer prevent the match. Usually, this means moving them outwards until they surround a copy of the induction hypothesis embedded in the rewritten induction conclusion. This rippling process is illustrated in Figure 1.

$$
\begin{aligned}
len(t <> l) = len(t) + len(l) \vdash & \\
len(\, h :: t \;<> l) &= len(\, h :: t\,) + len(l) \\
len(\, h :: t <> l\,) &= s(len(t)) \;+ len(l) \\
s(len(t <> l)) &= s(len(t) + len(l)) \\
len(t <> l) &= len(t) + len(l)
\end{aligned}
$$

Legend:

infix list constructor	$<>$	infix append
successor	len	length of list

The first line is the induction hypothesis; the other lines are the induction conclusion in successive stages of rewriting. In BARNACLE *wave-fronts are shown in red, but in this paper we have surrounded them in a grey box as a monochrome approximation. The skeleton is in normal mathematical font. The bits of skeleton which appear inside wave-fronts are called* wave-holes. *Note that with each rewriting the wave-fronts move outwards, then finally disappear to give an exact copy of the induction hypothesis. Fertilization will now complete the step case.*

Figure 1: An Example of Rippling

The rippling tactic is implemented as successive applications of a special kind of rewrite rule that we call *wave-rules*. The wave-rules are also annotated with wave-fronts. The defining properties of wave-rules are the preservation of the skeleton and the desirable movement of the wave-front (Basin & Walsh, 1994). Each wave-rule is responsible for moving the wave-fronts a little bit in the desired direction. To apply a wave-rule the left hand side is matched to a subexpression in the current goal and this subexpression is replaced by the instantiated right hand side. In this match any annotations occurring in the rule must also occur in the goal. This extra condition significantly reduces the number of rules that apply and hence the amount of search. Examples of wave-rules are given in Figure 2. More details about rippling can be found in Bundy *et al.* (1993); Basin & Walsh (1994).

There are an infinite number of induction rules for most recursive data-structures: one for each well-ordering of the data-structure. An inductive theorem prover must choose among these and this is one of the major causes of the combinatorial explosion in inductive theorem proving.

[1] Where s is the successor function, *i.e.* $s(n) = n + 1$

$$
\begin{aligned}
H :: T <> L &\Rightarrow H :: T <> L \\
len(H :: T) &\Rightarrow s(len(T)) \\
s(X) + Y &\Rightarrow s(X + Y) \\
s(X) = s(Y) &\Rightarrow X = Y
\end{aligned}
\tag{1}
$$

One source of wave-rules is the step cases of recursive definitions (c.f. *the first three rules above), but they also come from associative and distributive laws, the replacement law of equality* (c.f. *last rule above) and many other sources. Note that, in each rule, the wave-fronts on the right hand side are further out than those on the left hand side, except for the last one, where they are absent on the right.*

Figure 2: Examples of Wave-Rules

CLAM makes the choice by looking ahead into the rippling process and choosing an induction rule which best facilitates rippling. We call this *ripple analysis*. The key observation is that each induction rule inserts a different wave-front around the induction variable in the induction conclusion. *CLAM* looks ahead to see which wave-rules are available to ripple which wave-fronts around which induction variables. It then chooses an induction which will maximise the chances of rippling succeeding (Bundy *et al*, 1989). Other search decisions, like generalising the conjecture, introducing a lemma, making a case split or instantiating an existential variable can also be governed by the need to facilitate rippling (Ireland & Bundy, 1994).

5 Why Interaction is Still Needed

Proof plans capture the common patterns in proofs. However, some proofs contain parts that are unique to that proof. Proof planning cannot be used to guide search through these unique parts of proofs. We can either bridge the gap with random search or we can allow human intervention. Either solution brings problems. Random search reintroduces the problem of the combinatorial explosion. Human intervention in semi-automatic proofs means that human users must become oriented in failed proofs, *i.e.* they must understand where the automatic prover got to and why it is stuck.

Our hypothesis is that proof planning can help with this orientation problem. With the aid of the current proof plan the state of the partial proof can be described to the user using the specification language of the tactics at various levels of detail. The tactics divide the proof into chunks; larger tactics into a few big chunks; their subtactics into sub-chunks. The user can navigate through this space to understand the current blockage and its context. The blockage in the proof plan can be explained in terms of the failed preconditions of the tactics which were expected to apply or the unexpected success of tactics that were not expected to apply. The user can sometimes override the prover to force or prevent the application of a tactic. The user might use the specification language to analyse the failed proof and, hence, discover a proof patch. The patch might be to apply further rules or tactics to bridge the gap between the effects of previous tactics and the preconditions needed by a currently inapplicable tactic. To test our hypothesis we have implemented the BARNACLE system.

6 General Overview of BARNACLE

BARNACLE is a window-based interface to the proof planning system *CLAM*. It is implemented using LPA Prolog and Windows 3.1 for 486 PC compatibles: it is hoped to have a version for SUN workstations running SICSTUS Prolog soon, and a version for the Apple Macintosh at some stage. All communication is via a menu-system and dialogue boxes. The user need not take part in the theorem proving process at all, if the theorem is one which can be proved automatically. However, in addition to this automatic mode BARNACLE also offers an interactive mode, so that theorems which are presently beyond complete automation may be tackled - often with a mere nudge from the user at the appropriate time, the remainder of the proof being carried out automatically. The central feature which makes this possible is the proof planning technique which gives structure to the proof and a high level language in which to discuss it.

BARNACLE improves upon existing theorem provers in the following respects.

In wholly automatic mode, where *CLAM* can prove the theorem unaided, its output is clearer than *CLAM*'s, and the user may be selective in what is studied.

2. In either mode, the user may see explanations of the workings of the theorem prover. This is an excellent way of introducing novices to the techniques of inductive theorem proving and to the terminology of proof planning systems.

3. In interactive mode, where *CLAM* could not prove the theorem unaided, the user may intervene: the use of explanations and other special features make this much easier than with traditional interactive theorem provers.

The user of BARNACLE may browse through and load existing theorems stored in its library, and create new entries. The output of BARNACLE is divided amongst different windows (Figure 3). The main window of interest to most users is the plan window, discussed in §7.2. Windows other than the plan window which remain visible during the planning of a proof are the current subgoal; and a rewrite window showing the current portion of the proof. The user may invoke other windows showing, for example, the rules currently stored in memory; the methods and submethods available; or details of the ripple analysis used in selecting an induction rule.

7 Display Features

7.1 Overview

It is important that all information potentially useful to the user be available. However, to provide it on the screen all at the same time, as some theorem provers do, would be overwhelming. Different kinds of information are shown in different windows, and the theorem prover automatically brings the most current relevant information into focus during the course of a planning session. Information which is not deemed to be in focus is hidden or iconized, but can be revealed under user control if desired. The following sections describe the most important display facilities of BARNACLE.

7.2 Plans

The user may see the plan being gradually built up in the plan window as shown in Figure 4. The amount of detail given as default is sufficient for the user to get a broad overview of the proof, and check that everything is going smoothly. However, more detail may be required and the user may readily get this in two respects at any stage.

1. The user may see the subgoal to which a method was applied by clicking the appropriate buttons. Clicking on different nodes of the tree will bring up the appropriate subgoal in turn and hence show the user the intermediate steps of the proof.

2. The user may expand any node into its component submethods to see the proof in more detail.

The second of these needs further explanation. *CLAM* uses a hierarchy of methods and submethods. For example, the *basecase* method consists of applying one (or both) or two submethods: *symbolic evaluation* and *elementary*. *Elementary* may be applied to subgoals such as $e = e$ and is a *terminating* method (it finishes off its branch of the tree). *Symbolic evaluation*, like *basecase*, employs its own submethods, which are often applied iteratively until no more apply, as for example in:

$$\begin{array}{lll} 0.(y+z) = 0.y + 0.z & \Rightarrow_{times1} & 0 = 0.y + 0.z \quad \text{by eval\_def} \\ & \Rightarrow_{times1} & 0 = 0 + 0.z \quad \text{by eval\_def} \\ & \Rightarrow_{times1} & 0 = 0 + 0 \quad \text{by eval\_def} \\ & \Rightarrow_{plus1} & 0 = 0 \quad \text{by eval\_def} \end{array}$$

Here *times*1 and *plus*1 are the kind of rules that the user expects to find used in the base cases of inductive proofs: they often arise from the base cases of definitions, as here:

$$\begin{array}{rrcl} times1 & 0.X & \Rightarrow & 0 \\ plus1 & 0+X & \Rightarrow & X \end{array}$$

Note that the application of *elementary* to the final subgoal $0 = 0$ will now finish the proof. Clicking on *basecase* in the plan tree will expand it as shown in Figure 5.

7.3 Display of wave-fronts and other annotations

BARNACLE displays the annotations introduced in §4, thus showing the user the progress being made towards the goal. The example in §4 showed the annotations gradually moving outwards until they finally disappeared altogether. This does not always happen, although it is most desirable when it does, as explained in §8.3. The display of wave-fronts and other annotations is instrumental in enabling a user to take control of the proof if this suddenly becomes necessary.

BARNACLE displays these annotations in the rewriting window, giving the following benefits:

We can more easily see and understand the process of matching the rule and a subexpression of the subgoal: this feature proved particularly useful for novices.

2. The movement of the annotations from one line to the next graphically shows the progress of the proof: users have said that "it really does seem to 'ripple' ".

3. We shall see in §8.3 how the annotations can assist the user in assisting the theorem prover when the theorem prover gets "stuck".

7.4 Explanations

Theorem proving systems which give a blow-by-blow running commentary are hard to follow and may overwhelm the user. Traditionally, if the commentary is needed, for example to find out where the system has come unstuck, it is usually best saved and studied off-line. We have tried to avoid creating a system with two modes: "verbose" and "off", but instead allow the user to make use of the proof planning methodology which enables the user to set the system to give timely explanations of particular features. When used in conjunction with user veto (§8.2), this is an effective way of allowing human and automaton to work together using the strengths of each to best advantage. Humans are good at spotting bad moves, and also short cuts. We deal with the former in §8.2 and the latter in §8.3.

However, to make effective use of its human users, BARNACLE must keep them conversant with what is going on. As intimated in §5, the feature of *CLAM* which allows this is the specification of tactics using preconditions written in a meta-logic: moreover, each predicate of BARNACLE's

meta-logic corresponds to a high level, human-sensible property and is linked to a template for some explanatory text.

Given a subgoal, BARNACLE attempts to find an applicable method. This is a method whose preconditions are all satisfied by the current subgoal. Left to itself, (*i.e.* in wholly automatic mode) BARNACLE tries each available method in a predetermined order, until it finds one which is applicable. It does this by testing each precondition in order, carrying on as long as the preconditions succeed, but moving on to the next available method (if any) if a failing precondition is encountered. In wholly automatic mode, a method is selected only if all its preconditions succeed.

As intimated above, each precondition is linked to an explanation: in fact two explanations, to allow us to explain failure as well as success. To trace the successful use of a given method, the user may switch on an explanation for the method, to be given in the case of succeeding preconditions only. This is useful for novices learning inductive theorem proving techniques, but is less so for experienced users of the tool, who are more likely to request explanations in case of failure. Moreover, explanations of failure are generally only useful when given in context: *i.e.* in terms of what prior preconditions succeeded before the failing precondition was tested. Explanations in BARNACLE therefore take the form shown in Figure 6.

Preconditions are not all of equal status. Some are hard and fast conditions: for example, to be applicable the ripple method must have a suitable matching rule available. Without such a rule, to apply the method would be nonsense. On the other hand, other preconditions are more heuristic in nature. Many of the preconditions associated with choosing an induction schema are rules of thumb and if satisfied imply only that there is a strong chance of success: but if they are not satisfied then success may still result. Used in conjunction with the user's power of veto (§8.2) this is a powerful feature of the system, but only if the user understands the reasoning behind the choice of method and is effectively oriented in the proof, understanding the context in which these tests are being carried out.

8 Interaction

8.1 Editing and Recording Proofs

Because BARNACLE is window-based, it may be used in conjunction with other window software, such as word processors. For example, a record may be kept of the tree representation of the plan; details of the ripple analysis may be saved; or proofs may be cut and pasted into documents. This is useful when attempting proofs whose details may be of interest to a wider audience, or for students learning inductive techniques.

In the context of proving more difficult theorems, it may be useful for users to perform rewriting themselves and have this verified by the system. We give an example here. Many theorems cannot be proved in their original formulation but a generalized form of the theorem may be more readily provable. It surprises many people to find that whilst the trivial theorem

$$x + (y + z) = (x + y) + z \tag{2}$$

is easy to prove, the specialized version

$$x + (y + x) = (x + y) + x \tag{3}$$

is not. If this theorem is attempted in automatic mode by BARNACLE, the user soon spots that the proof attempt is doomed to failure. The proof technique needed here is that of generalizing apart the occurrences of x, *i.e.* rewriting (3) to (2). A body of work exists on this technique but it has not proved possible to easily incorporate it into the release version of *CLAM*. BARNACLE proved much more amenable to this technique, however, as the user can interact directly with the conjecture using the usual simple windows editing techniques: BARNACLE reads the new conjecture and verifies that it truly represents a generalization apart of the original.

8.2 User Veto

CLAM encapsulates a very powerful yet flexible strategy for proving theorems. It is much better than the average human at searching and matching appropriate rules and calculating complex heuristics for ripple analysis. Nevertheless, there are times when human intervention is necessary. Humans are better, in general, in spotting steps which are almost certainly doomed to failure. There are several categories to consider where human intervention may prove timely.

Choosing an appropriate induction rule.

2. Avoiding overgeneralization.

3. Overriding the theorem prover's fixed preference for one method over another.

We consider each of these in turn.

Ripple analysis incorporates a collection of heuristics. These are not guaranteed to choose the correct induction schema every time, although they often do, and are unlikely to improve in the near future. Indeed, one of the benefits of BARNACLE is that it is easy to experiment with different heuristics and different choices in the development of more accurate and more powerful heuristics. The ideal human-automaton team lets the automaton make the initial choice, which involves much search and computation, but lets humans use their mathematical experience in exercising right of veto over obviously bad choices as soon as this becomes apparent.

Over-generalization is another trap for the unwary theorem prover, as a non-theorem could result. The wholly automatic theorem prover can spent a lot of time chasing a proof of a conjecture which to the human user is clearly false. Ideally a disprover should come into play following generalization but in practice, due to incompleteness it is an open question as to how to divide time and resources between attempting a proof and seeking a counterexample. This is another area where humans often have the edge over automata.

Finally, the standard order in which *CLAM* attempts methods, whilst ideal for many theorems, occasionally comes unstuck. For example, examples exist where symbolic evaluation is performed as a first step in the proof when induction is necessary and *vice versa*. If this leads to eventually failure and backtracking leads all the way back to this initial wrong step, then the right step will then be applied and a successful proof found. In practice, this may not happen in real time and indeed need never happen at all if an infinite branch is encountered. Timely intervention by the user avoids this situation.

User veto takes two forms:

The user may exercise right of veto over all methods chosen by BARNACLE.

2. The user may select certain method(s) only over which they wish to exercise right of veto.

The first of these is less useful and should be chosen (say by students and system developers) only if a detailed analysis is sought or in experimenting with different proofs. The second may be used for any or all of the methods intimated above which occasionally cause trouble, namely induction, generalization, and symbolic evaluation; and any other methods which are provided (this can be customized). If the user selects, for example, generalization from this menu, then each time BARNACLE wishes to apply the generalization method it will ask the user first. The subgoal which would result is displayed and a dialogue box allows the user to:

1. allow the use of the method;

2. veto the use of the method: BARNACLE must backtrack and find something else;

3. allow the method and switch off the power of veto from that point onwards; or

4. abort the proof attempt.

The third feature allows the user to switch off potentially tiresome messages once the danger point is deemed to be passed. A similar feature exists for explanations.

So far we have spoken of the user rejecting a method which BARNACLE has selected. We may also wish to exercise choice the other way. BARNACLE allows the user to override certain selected preconditions. These are all heuristic in nature: BARNACLE does not allow hard and fast conditions to be overridden. The user can bring up a menu of methods and associated overridable preconditions, and select one or more of these before starting to plan a proof. If a precondition selected in this way fails at any stage of the proof, the user will be allowed to override it.

8.3 Use of Lemmas

Possibly the feature of this interface which renders it most amenable to developing human-like proofs is the ability it gives the user to plug in known lemmas on line, without interrupting the proof planning process.

In the example of §4 we saw that rippling continued with the successive application of wave-rules until all wave-fronts disappeared and we could see a copy of the induction hypothesis which was then used to finish the proof. However, this is often not possible, and other methods must then be applied at the point where rippling is blocked. Sometimes it is relatively simple to complete the proof, but we often have a nested induction to perform. The proofs can grow quite complex, and sometimes unnecessarily so — as when a simple lemma, if present, could have been used at the appropriate point.

A user is often capable of spotting that a commonly known lemma could be used in the proof to make further progress, and furthermore this lemma could be one already proved and resident in the library. BARNACLE loads only a minimum of definitions *etc.* unless otherwise prompted, and in general this is quite a good thing given that extra lemmas can lead to more search and slow down the overall planning process. However, it is useful, once the need for such a lemma becomes apparent, if the user can invoke such a lemma, and have BARNACLE load it in and make use of it in the proof. In other systems, additional lemmas may only be loaded by interrupting the proof and beginning again. Here the user is prompted in places where BARNACLE knows there is a fair chance that a lemma could be useful, *i.e.* where rippling is blocked and strong fertilization cannot take place. The lemma is loaded and the plan proceeds after this small interruption.

In the context of blocked rippling, the annotations displayed by BARNACLE are most valuable. For instance, rippling of the goal:

$$x + s(y + z) = s(y + (x + z))$$

will get stuck if there is no wave-rule applicable to the left-hand side[2]. The user can infer the pattern of the missing wave-rule by examining the wave-annotation of the stuck goal. It is:

$$+ s(V \Rightarrow F(x + V)$$

Note that the wave annotation suggests which bit of the goal to generalise to the variable V, namely the contents of the wave-hole. It is also suggests the pattern of the right-hand side of the wave-rule. The wave-rule:

$$U + s(V) \Rightarrow s(U + V$$

unifies with this pattern. This missing wave-rule can then be proved, annotated[3] and loaded, and the ripple resumed. The recognition of the pattern is stronger when the wave annotations are presented in colour by BARNACLE, rather than the monochrome to which we are restricted

[2] Note that wave-rule (1) does not match.

[3] In this case, these first two steps are unnecessary, because the rule is already in BARNACLE's library, but not loaded.

in this paper. A possible extension to BARNACLE would be to get it to work out the required pattern, and display it to the user and ask for suggestions as to wave-rules that might fit the pattern.

9 Evaluation

9.1 Empirical Evidence

The BARNACLE system is available on application to the first author, and needs a PC running Windows 3.1 and LPA Prolog for Windows version 2.6 or higher. At the time of writing, the system is being evaluated on a number of fronts. Full analysis of the data gathered will be complete by June 1995, but meanwhile the early indications are encouraging. Studies being carried out or imminent include:

1. experienced and novice users' evaluation of the representation of plans by hierarchical tree structures;
2. the helpfulness of ripple annotations in teaching novices how to do inductive proofs; and
3. whether user interaction via lemma introduction improves either user performance or the quality of proofs found (in terms of shortest proofs or proof structure).

9.2 Usefulness of the Plan Hierarchy

Users of proof editors often complain of getting "lost" in a proof. This tendency increases with the complexity of the proof. For instance, it is easy to lose track of which step case of a nested induction to return to, or whether we have completed all cases of a case analysis. We hoped that a multi-level plan hierarchy for BARNACLE will help to solve this problem. The proof plan itself is of central importance in user interaction in orienting the user within the proof attempt. We therefore assessed the needs of current main users of the *CLAM* system, namely members of the Mathematical Reasoning Group at Edinburgh University.

Most, although not all, felt that the previous representation of the plan was difficult to understand even for small proofs, and for large ones was unhelpful. Particularly frustrating was the fixed level of detail presented - this could be set by the user prior to the proof attempt but typically provided either too much detail, or too little. As one user put it "You either get swamped with information or else you can't see the crucial bit, depending on what formatting option is chosen." The optimal representation appeared to be a hierarchical tree structure, where contracting and expanding each node was at the behest of the user, with methods linked to the particular subgoal they were applied to.

Other desirable features were seen to be the use of highlighting or colour to improve readability and focus on important aspects, such as terminating methods and wave-fronts within subgoals. It was important to have a zoom facility so that the overall structure could still be referred to even if the tree became large. For document preparation, for example in order to write reports on proof attempts or papers containing examples of the techniques, it was seen as important that printing facilities were available linked to the planner. These criticisms informed the design of BARNACLE.

BARNACLE has been evaluated for "look-and-feel" by the original target group. Reaction was very favourable, and initial feedback and additional suggestions pointed out by the group are being readily incorporated. The new interface makes it much easier to understand what is going on in the proof and allows the user to concentrate on the important aspects of theorem proving rather than interpreting the output.

A reservation which some of the evaluees expressed was whether the implementation would "scale up". We return to this question in §10. It should be pointed out that the previous representation was particularly bad for large plans with a lot of nesting.

A true evaluation needs to be more objective, in terms of performance. We plan further studies to take objective measures of users comparing the new representation compared with the old in terms of number of successful proof attempts, time taken to complete proofs, and post-hoc understanding of the structure of proofs.

9.3 Usefulness of Annotations

Indications so far are that complete novices find great difficulty even in matching subexpressions, as in the example given in §4. We are currently examining the proficiency of novice students in carrying out proofs of various levels of difficulty:

1. after a standard introduction of proof by induction;
2. after learning about proof planning and rippling; and
3. after hands-on experience with BARNACLE.

Our evaluation, due to be completed in June 1995, has the following methodology.

Around 80 students are involved, all taking the same course (in formal methods, on a computing degree at Napier University) but with a wide range of backgrounds and experience. Their concern with inductive theorem proving arises from its link with the verification of recursive programs rather than any particular interest in mathematics. We have collated as much background information as possible, in terms of qualifications, performance in previous mathematics courses, age, work experience, and attitude to mathematics.

All students were collectively given a standard presentation of the method of proof by mathematical induction, using a traditional lecture format. Subsequently they all carried out exercises designed to test their understanding of this and related material, notably the use of recursive definitions and the ability to match definitions to subexpressions.

Following this initial exposure the students working in smaller units are being given the same activities (for ethical reasons) but in different orders (for the purpose of evaluation). These activities comprise learning the "rippling story" of inductive theorem proving, and hands-on experience of using BARNACLE. The students are given exercises to test their understanding at various stages. As a control, some students are at first simply given more exercises in induction, to allow for the effect of other factors thought to improve performance, such as working in smaller tutorial groups, repetition, and assimilation of ideas with the passage of time. Others are first given instruction in the proof planning idea and rippling, and others have this and a demonstration of BARNACLE, which they are then free to use, although they must carry out some exercises without its assistance. Finally, all students will sit an examination in June. The full set of results will be analysed using parametric and non-parametric methods designed to factor out confounding factors and to investigate not simply the performance of all students over the three groups but also whether BARNACLE is of special value to students with poor prior understanding of mathematical techniques.

9.4 Use of Lemmas in Proofs

The study in §9.3 incorporates an investigation into whether novices can be successfully prompted to use lemmas in structuring their proofs. We plan further experiments to test this feature with experienced users and more difficult proofs, some of which cannot be proved automatically without the provision of lemmas. If successful, this would represent a major advance in the power and usefulness of theorem proving systems, and bring them within the range of less skilled users.

The methodology here will consist of supplying a number of theorems to be proved, in random order. Some can be proved only by introducing lemmas, others can be proved without but the use of lemmas admit clearer, shorter proofs, whilst others do not benefit from the use of lemmas at all. One set of users will attempt these proofs using *CLAM* without the BARNACLE interface, whilst others will use BARNACLE. We shall compare performance by logging interaction with the systems, the proofs found, and the time taken to find them.

10 Current Limitations

10.1 Default level of detail

Our hypothesis is that the most sensible representation of a plan is one which by default operates at a high level, but which allowed for expansion into more detail, as required. This hypothesis has be confirmed by our initial investigations. The question remains as to how high the default level should be. At present we represent induction as a node labelled with the particular induction schema chosen which has one or more branches corresponding to the base cases, and one or more branches corresponding to the step cases.

The fear was expressed (see §9.2) as to whether the representation will cope with large scale proofs. This is not an easy issue in any representation. There are two options both worth trying if this becomes a problem.

One option is that we can in fact go up one further level, and represent induction as one node incorporating the whole induction strategy, in other words base and step cases would be incorporated within the one induction node, rather than as separate branches of a tree. Of course, at the click of a button this node could be expanded into the tree form, as with any other node. Thus, for example, a proof consisting of an induction with a base and a step case, the step case of which was followed by an induction which then terminated, would be represented as just two nodes comprising the chosen induction schemata, and the details of base and step cases would be hidden unless invoked by the user.

The other option would be to allow the user to hive off part of the proof by selecting a new root for the tree at the place of current focus. When this was completed, the user could return to the original tree.

10.2 Multiple expansion of nodes

The implementation currently being evaluated allows only one node of the tree to be expanded at any one time. Some users expressed the desire to be able to leave several nodes expanded at once. Further investigation will determine whether allowing several expansions in one tree is desirable. It seems best to give the user the choice to allow this, although redrawing complete trees could be time-consuming and we could end up back in the previous situation where the user is swamped with too much detail.

As with the level of detail provided as default, it seems that the user should be able to set the defaults of the system, one set being seen as more desirable for small proofs and novice users, and another for large proofs and more experienced users.

11 Related Work

There are a large number of interactive theorem proving systems. Below we compare BARNACLE with a representative sample of these.

Some of the best known and best engineered interactive provers belong to the LCF family (Gordon *et al*, 1979), *e.g.* LCF, HOL, Isabelle and NuPRL. We take NuPRL as representative of this family. NuPRL (Constable *et al*, 1986) allows definitions, theorems, and proofs to be created in a window-based system using a mouse interface. Commands may be entered in one window to go into edit mode, access the library, and to execute steps of the proof. Definitions, theorems, and tactics may be stored in the library and viewed by executing the appropriate commands. There is a text editor, and a proof editor. The proof editor window displays the current subgoal. The user may move around the proof and may refine a step by giving a rule which can be applied to that step. Assistance to the user consists of checking that the rule is applicable and if so the new subgoals are computed. Detailed prior knowledge of NuPRL's Type Theory is of some assistance to users in learning how to use the system and it is wholly interactive — the system cannot suggest which rule should be tried next. Interaction is at the

object-level, except that users can execute tactics instead of individual rules and the proof is displayed in tactic sized chunks.

NQTHM (Boyer & Moore, 1979) is generally considered the state of the art inductive prover, and has been widely used for practical verification problems. Although, unlike NuPRL, it is an automated theorem prover rather than merely a proof editor/checker, experience is needed in its use in order to prove non-trivial theorems (Kaufmann, 1990). Its output, although useful, is verbose and often deliberation is needed offline in order to set up the lemmas needed by the proof, in contrast to BARNACLE (see §8.3 above). Moreover, the output comes out in a steady stream, and the user, employing visual alacrity, manual dexterity, and experience, must spot where a lemma could have been usefully employed and abort the system — the system does not stop at strategic places as would BARNACLE. The output describes the object-level proof and there is no chunking.

INKA (Biundo *et al*, 1986) is, like BARNACLE, a window-based system and provides windows for editing, for viewing library objects, and for displaying proofs. The conjecture to be proved is negated and a contradiction is sought. It is semi-automatic and displays inductive proofs using tree-structures whose nodes can be expanded to show different information — in fact BARNACLE has borrowed some of its representational designs from INKA. It is semi-automatic in that, for example, it will not spontaneously perform a nested induction but will show the node as incomplete (completed nodes are shown in bold boxes). The user is then given a choice of options, including one to allow the theorem prover to proceed automatically. Its display facilities are readily understood, although it cannot provide explanations for steps nor advice on what to do next. We have worked closely with the INKA developers on rippling and related techniques, and this is reflected in the relative closeness of our systems.

All these systems give low-level descriptions of the proof. None of them exploits the advantages of proof planning to give explanations of the proof at a higher-level. They exploit tactic-like ideas to chunk the proof, but only INKA displays the proof hierarchically in the way that BARNACLE does.

12 Further Work and Conclusions

12.1 The next steps

§10 described limitations of the current implementation and work on investigating these and on incorporating user feedback is ongoing. This section describes more interesting developments in co-operative theorem proving which BARNACLE makes possible. Further work planned in the near future comprises

1. Improving user interaction with the planning process.
2. Improving the quality of advice given to the user.

12.2 User intervention

Currently, the user may intervene in the planning process only by exercising a right of veto over a method, or in overriding preconditions. It is a simple amendment to the planning process to allow the user to see all applicable methods at a node, and exercise choice between them. The use of a best-first planner (Manning *et al*, 1993) will additionally present the user with heuristic measures of the relative merits of all methods applicable at that point.

However, the best mode of intervention is where the user may interact directly with an incomplete plan tree. For example, the user should ideally be able not just to inspect a node during the planning process (which is now possible) but also should be able to interrupt the planning process to undo a node, or switch attention from one incomplete branch to another. This has long been seen as desirable, but a persistent representation of the plan with which the user could communicate via simple commands was an essential prerequisite. We hope to extend

the work done on representing plans so that the user can communicate with the planner using mouse clicks and choose from a number of options: such as seeing what methods are available at a node, choosing an applicable method, undoing or redoing a node, and suspending work on a branch.

12.3 Advising the user

At present, BARNACLE, like *CLAM*, will stop at the first failing precondition of a method as explained in §7.4. Critics (Ireland & Bundy, 1994) work by evaluating *all* preconditions. If all succeed, then the procedure is the same as for BARNACLE. However, if one or more preconditions fail, a critic for that method may exist and in this case automatically proposes a patch. The patch suggested will depend on the combination of succeeding and failing preconditions. Patches will not normally be suggested if all the preconditions fail, but if at least one succeeds then this may suggest the next action. For example, to apply the *wave* submethod of *rippling* there must be a wave-front as a subterm of the current subgoal, and also a wave-rule which matches it. If the first of these conditions is true but not the other, a critic for the wave submethod can both propose and prove a lemma to be used in a proof, using reasoning similar to that described at the end of §8.3.

At present, only an experimental implementation of *CLAM* incorporating critics has been built, but if BARNACLE could incorporate critics as *advisors* to the user then the scope and power of both systems could be increased. Critics could also be used under user control to generalize the conjecture, propose a case split, or instantiate an existential variable, thus avoiding the fruitless search which sometimes results during some proof attempts. These patches, along with the speculation of lemmas, are difficult to automate completely, but the reasoning produced by the critics may be exactly what the user needs to make an informed decision as to what to do next. For example, we can imagine the system explaining that rippling cannot take place because there is no suitable wave-rule available (as it already does) but rather than merely displaying the current subgoal with its annotations and leaving the user to pattern match with an appropriate lemma, the system itself could propose the form of the lemma and let the user decide whether to seek such a lemma or if none exists in the library whether to allow the theorem prover a free rein in attempting a proof of such a lemma as a subproof of the main theorem. At present, work on critics and work on co-operative theorem proving could be seen as alternatives, one seeking complete automation and the other some interaction, but they should be seen as complementary.

12.4 Conclusion

The BARNACLE interface appears to increase the power of existing proof planning systems. It makes them easier to learn how to use, and provides a vehicle for introducing users to some powerful techniques in theorem proving and allowing many proofs in this undecidable domain to be almost completely automated, even though complete automation is impossible. It provides a conduit for the most recent work in theorem proving to be released into the field at the earliest opportunity and prove beneficial to practitioners of both mathematics and programming as well as students and other interested parties.

Proof planning makes new kinds of user interaction possible. It enables proofs to be displayed in a multi-level hierarchy. This dramatically reduces the cognitive load on users and permits them to navigate around a proof, viewing it in more or less detail. It provides high-level explanations of the roles of the chunks within the proof. These explanations assist the analysis of a failed proof attempt. It also provides a high-level language for interaction with the system. This can be used for interactive patching of partial proofs.

References

Basin, D.A. and Walsh, T. (1994). Annotated rewriting in inductive theorem proving. Technical report, MPI, Submitted to JAR.

Biundo, S., Hummel, B., Hutter, D. and Walther, C. (1986). The Karlsruhe induction theorem proving system. In Siekmann, Joerg, (ed.), *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag. Springer Lecture Notes in Computer Science No. 230.

Boyer, R.S. and Moore, J.S. (1979). *A Computational Logic.* Academic Press, ACM monograph series.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag. Longer version available from Edinburgh as DAI Research Paper No. 349.

Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A. and Stevens, A. (1989). A rational reconstruction and extension of recursion analysis. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann. Also available from Edinburgh as DAI Research Paper 419.

Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567.

Char, B.W. et al. (1993). *Maple V library reference manual.* Springer-Verlag.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

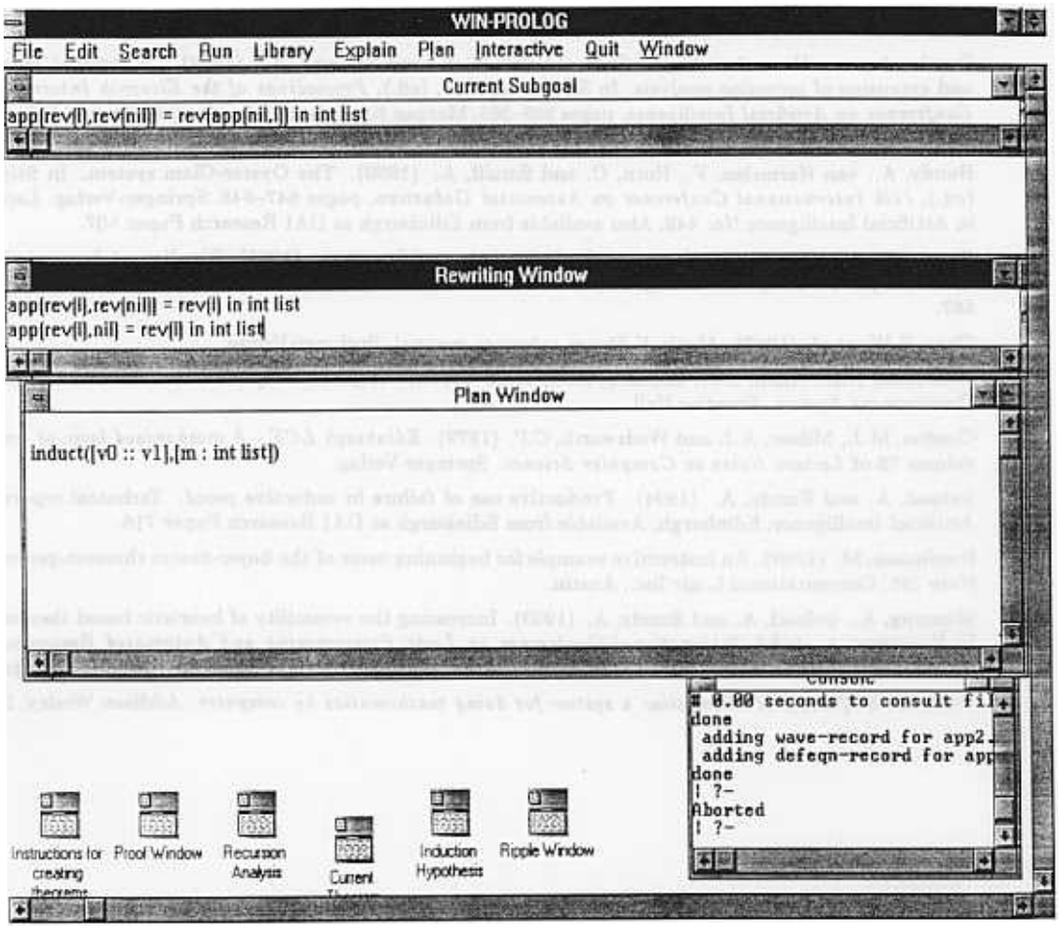
Gordon, M.J., Milner, A.J. and Wadsworth, C.P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag.

Ireland, A. and Bundy, A. (1994). Productive use of failure in inductive proof. Technical report, Dept. of Artificial Intelligence, Edinburgh, Available from Edinburgh as DAI Research Paper 716.

Kaufmann, M. (1990). An instructive example for beginning users of the boyer-moore theorem-prover. Internal Note 185, Computational Logic Inc., Austin.

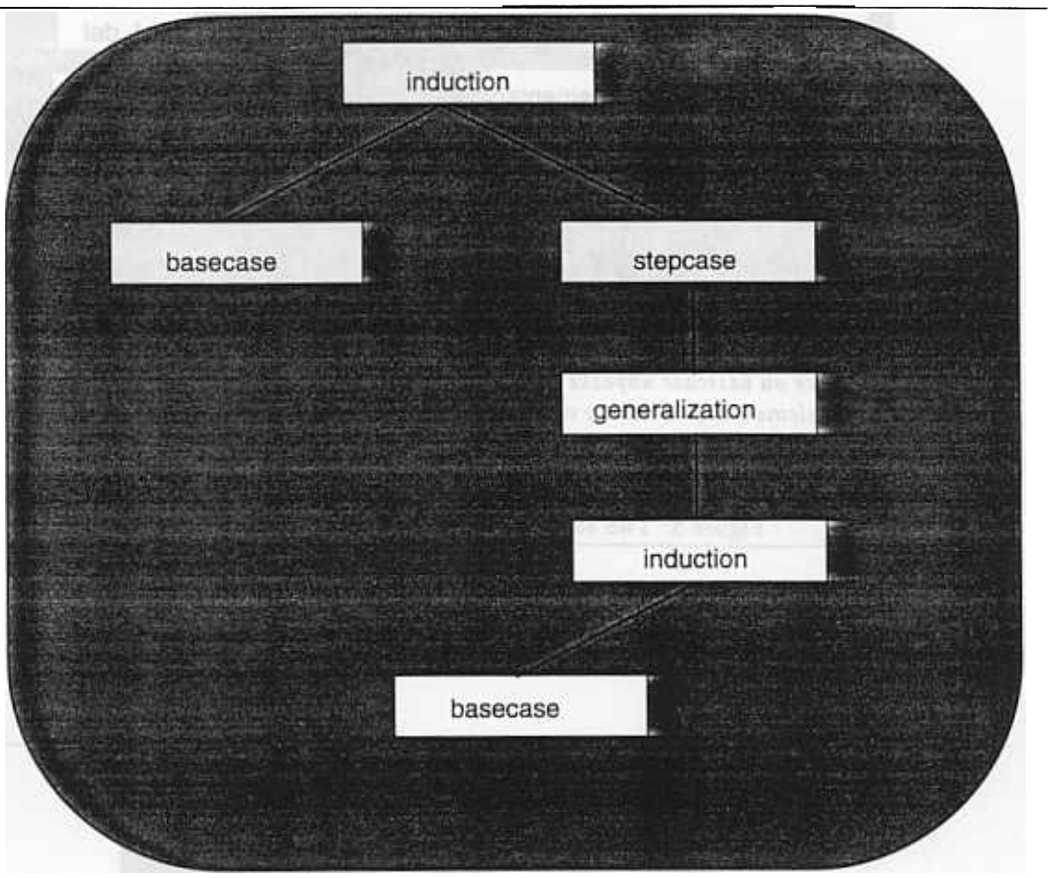
Manning, A., Ireland, A. and Bundy, A. (1993). Increasing the versatility of heuristic based theorem provers. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning - LPAR 93, St. Petersburg*, number 698 in Lecture Notes in Artificial Intelligence, pages pp 194–204. Springer-Verlag.

Wolfram, S. (1991). *Mathematica: a system for doing mathematics by computer.* Addison Wesley, 2 edition.



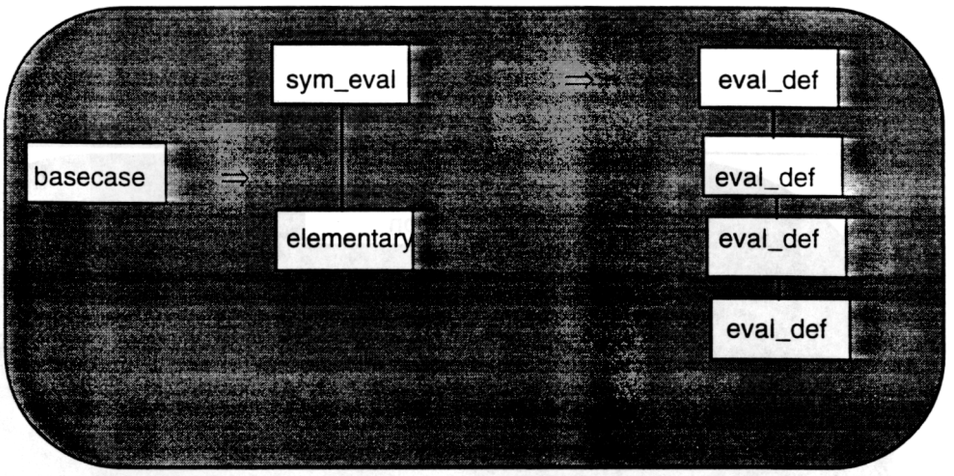
Four windows are open: current subgoal; rewriting; plan and consult. The current subgoal window shows the formula that BARNACLE *is currently trying to prove. The rewriting window shows the before and after state of the expression being rewritten. The plan window shows the plan which has been constructed so far. The consult window is provided by default by the* LPA *Prolog system and is not used. Other windows are shown iconised at the bottom of the screen.*

Figure 3: The BARNACLE Interface



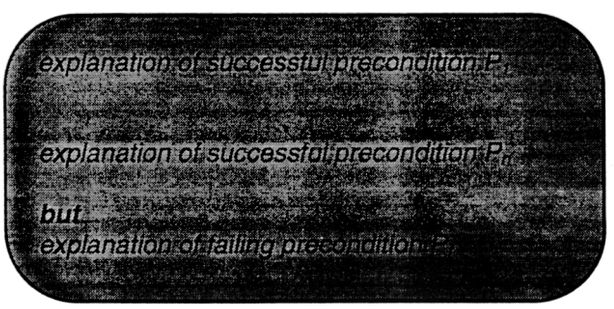
The theorem prover has applied induction to the original conjecture, and subsequently basecase submethods are applied to the first subgoal which arises. To the other subgoal (note the branching structure of the proof) the theorem prover applies stepcase submethods, followed by a generalization of the resulting subgoal, and another induction, and so forth.

Figure 4: Partially completed proof plan



One click on basecase unpacks it into an application of symbolic evaluation followed by one of elementary. Clicking on symbolic evaluation unpacks it into four applications of evaluate definitions.

Figure 5: Two successive expansions of the basecase method



Each CLAM method has a conjunction of preconditions. Normally all of these must succeed for the method to be applicable. When failure reporting is switched on for a method, BARNACLE *will give an explanation of the first failing precondition preceded by an explanation of the previous n succeeding ones.*

Figure 6: Form of explanations